TD/TP 5

Classes, les bases

Objectifs, méthodes

Enfin vous arrivez au vif du sujet : implémenter vos propres classes C++ correspondant aux classes d'un modèle objet adapté à une application. La traduction en C++ des associations UML entre entités posent des problèmes spécifiques qu'on abordera au cours 6 et TD/TP 6. Pour l'instant on va se concentrer sur l'aspect le plus simples et le « plus bas niveau » de la relation entre classes : la composition entre types valeurs (*value types*). Pour se faire nous reprendrons la struct Sphere du TD/TP 4 et nous la convertirons en classe, en la dotant d'une structure hiérarchique de composition. **On veut une application avec les mêmes fonctionnalités que les exos 2 à 7 du TD/TP 4, mais en utilisant des classes et des méthodes à la place de la struct et des fonctions (sous-programmes)**

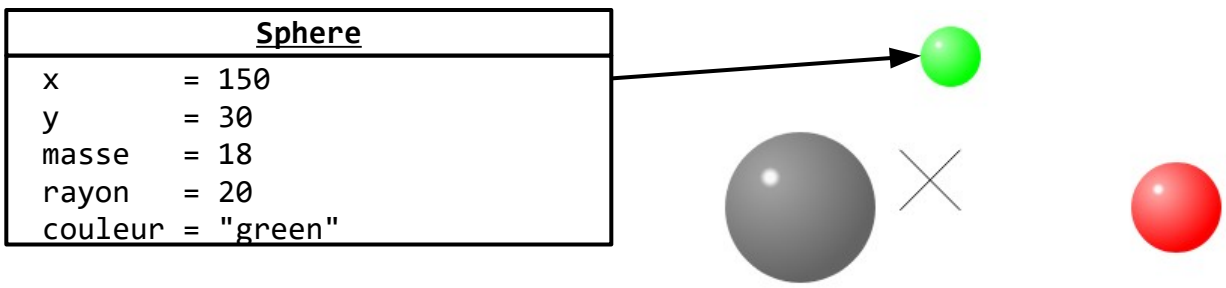
Pour aborder confortablement les 3H de ce TD/TP il est indispensable d'avoir bien suivi et probablement relu le cours 5 sur les classes en C++

Les concepts du cours couverts par ce TP seront

- **`classes simples`** avec des attributs types « élémentaires »
- **`constructeurs`** pour des classes simples
- **`méthodes`** pour des classes simples
- **`opérateurs`** pour des classes simples
- **`diagramme de classes et composition`** composants/composite chapitre H cours 5
- **`classe composite`** avec des attributs de type classe composante
- **`constructeurs`** pour une classe composite
- **`méthodes`** pour une classe composite
- **`new/delete`** pour ajouter/enlever dynamiquement des objets composites d'une collection

1. Structure hiérarchique de composition : réflexion, modèles UML

Pour rappel au TD/TP 4 on étudiait la réalisation d'une application de gestion d'objets sphériques massifs dans le plan (2D) et du calcul de leur barycentre. On voulait en particulier pouvoir illustrer graphiquement le centre de gravité de systèmes de corps astronomiques (planètes, satellites, exoplanètes, étoiles à neutrons...) réels ou imaginaires. Pour **grouper les données** d'un même objet céleste nous utilisions une simple **`struct Sphere`** ce qui techniquement en C++ est une classe avec des attributs publiques, et nous n'avions pas de méthodes mais des fonctions prenant des (références) à des objets de type Sphere en paramètre. Pas très « orienté objet » ...



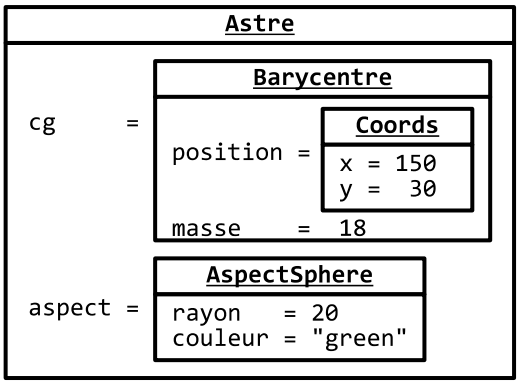
Le diagramme page précédente est un **diagramme d'objets** reprenant les conventions des diagrammes d'objets du TD/TP 1. Avec les diagrammes d'objets spécifiquement je me permets de prendre quelques libertés par rapport à la norme UML en me rapprochant de « schémas mémoire ». Pour rappel dans un diagramme d'objets le nom des classes est souligné.

Pour les **diagramme de classes** nous essayons de suivre la norme à la lettre. Dans les diagrammes de classes le nom des classes n'est pas souligné, **et bien sûr dans un diagramme de classes aucune valeur concrète d'instance particulière n'apparaît**. Aucune confusion n'est donc possible. Nous allons maintenant discuter de l'art et la manière d'accommoder à la sauce objet les **données** de l'instance « boule verte » : concrètement {150, 30, 18, 20, "green"}

Ce simple « agrégat de données », qui ne fait sens que pris ensemble (la valeur 30 à elle seule, isolée de son contexte, est inexploitable) peut adopter bien des formes selon la façon dont nous allons le **structurer**. C'est notre travail de concepteur d'envisager les différentes options :

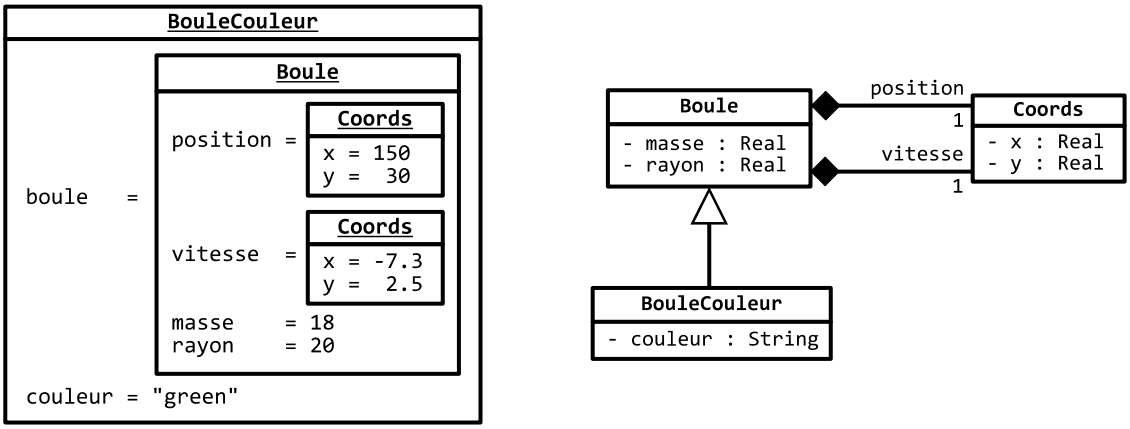
- Il peut rester « à plat » avec les données dans un seul bloc. En gros on reprend la déclaration du TD/TP 4, on remplace `struct Sphere` par `class Sphere` et voilà ! Pourquoi pas ! Sauf que *si vous testez* vous allez voir que ça ne passe pas au compilateur : les attributs se retrouvent privés donc inaccessibles par les fonctions qui traitaient des paramètres de type Sphere. Il va falloir **transformer ces fonctions en méthodes**. *Plus tard* !
  Car avant de se lancer dans un gros travail de mécanique de code réfléchissons : est-il vraiment satisfaisant d'avoir **au même niveau** un attribut y, qui ne fait vraiment sens qu'avec x, et un attribut couleur, qui n'a de sens que lors du dessin final mais ne jouera aucun rôle durant les calculs géométriques ? En informatique on aime bien séparer les aspects traitements des aspects présentation... D'autre part il est probable que si nous faisons de la mécanique céleste (en 2D dans le plan de l'écliptique, par soucis de simplification) nous aurons biens d'autres occasions d'utiliser des couples (x, y). Il serait scandaleux de devoir manipuler/recoder des opérations vectorielles spécifiquement pour des (x,y) de Sphère, pour des (x,y) de vecteur vitesse, pour des (x,y) de vecteur accélération, pour des (x,y) de foyers d'ellipses orbitales etc... On comprend qu'un type « vecteur 2D » se dégage dans l'analyse de ce genre d'application, et qu'on va vouloir exploiter ce type dans le type Sphere. Même si pour l'instant le CDC (un peu vague) ne nous parle que d'un seul type...

- Un développeur senior aura peut être l'envie de voir les choses de la façon suivante : en utilisant un type « vecteur 2D » (que nous appellerons Coords en référence au cours) pour la position des sphères, un type Barycentre pour grouper les données « **physiques** » et d'autre part un type spécifique pour encapsuler les aspects de **présentation** sous forme de sphères de couleur mais qui ne jouent aucun rôle dans la mécanique des calculs, qui s'appellerait AspectSphere par exemple.



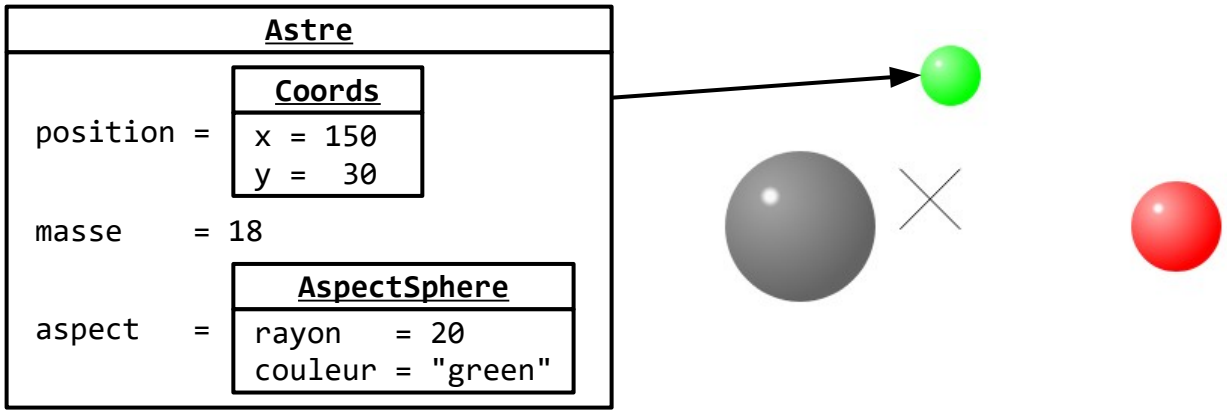
1.1 Faites le diagramme de classes UML correspondant à ce diagramme d'objets

- **Dans l'hypothèse où** nos sphères devraient être utilisées pour faire une simulation de billard, ou de pétanque, ou de modèle simplifié de mélanges de gaz mono-atomiques, donc des modèles dans lesquels non seulement la position et la masse des sphère est importante pour les calculs, mais aussi leur rayon (pour savoir quand elles sont en collision), alors la donnée **rayon** sortirait du cadre de la présentation (aspect) et rentrerait dans le cadre de la mécanique. Ne resterait que la seule **couleur** à la rubrique présentation, ce qui ne justifierait pas un type à part (à moins qu'on envisage d'avoir à ajouter plus tard d'autres données de présentation : texture, brillance, transparence...). Naturellement il faudrait aussi des données cinématiques (vecteur vitesse). Ci dessous à gauche un diagramme d'objets correspondant à cette **problématique différente** : **noter que le rayon se retrouve maintenant dans le même bloc de données que la masse**. Noter également que cette proposition à gauche sous forme de composition ne serait sans doute pas la solution choisie : dans ce cas précis on utiliserait plutôt un héritage car clairement une BouleCouleur **est** juste une Boule avec un truc en plus, ce qui correspondrait au diagramme de classes à droite.



1.2 Faites le diagramme de classes UML correspondant au diagramme d'objets à gauche
*Le diagramme de classes à droite est une alternative avec **héritage**, il est différent de celui demandé*

- Finalement, pour le problème qui nous concerne de modéliser de centres de gravité de corps céleste je vous propose un modèle un peu moins complexe que celui page précédente :



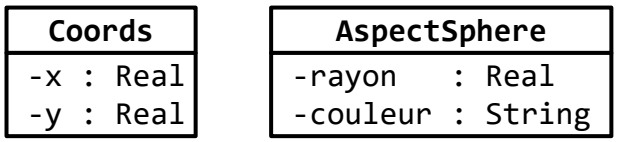
1.3 Faites le diagramme de classes UML correspondant à ce diagramme d'objets

C'est cette dernière structuration des données que nous adopterons pour coder ce TP. Elle résulte d'un ajustement entre les impératifs de la programmation objet (grouper des unités sémantiques et opérationnelles cohérentes), les limites du CDC implicite (travailler avec des corps célestes, pas des boules de bowling), une anticipation de futurs projets (on nous a parlé de maillages 2D comme fil conducteur, une classe « vecteur 2D » sera sûrement utile) et les besoins pédagogiques (pas trop de niveaux de composition, pas trop de classes).

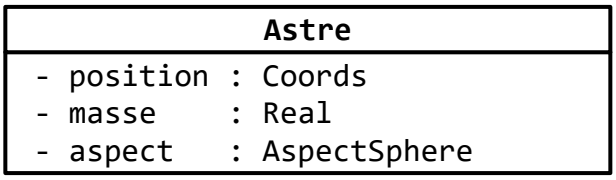
Sauter sur le clavier et entrer directement 5 attributs « en vrac » dans une classe, sans faire ce travail de réflexion, ce serait mal faire le travail. Même si on peut arriver dans un contexte particulier (prototypage, urgence à livrer, CDC très circonscrit) à la conclusion que, finalement, l'approche « la plus simple » de mettre tous les attributs ensemble au même niveau est la meilleure, il faut avoir envisagé des alternatives. Nous verrons qu'une structuration composant/composites

- impose immédiatement un travail supplémentaire : plus grande complexité, mise en place des classes composantes, *forwarding* des paramètres dans les constructeurs...
- permet un bénéfice sur le long terme :code conceptuellement plus propre, responsabilités limités de classes plus petites donc individuellement plus faciles à développer et tester, classes extensibles sans tout casser, classes ré-utilisables, code des couches intermédiaires plus expressif, meilleures abstractions…

Au final, ces classes composants que nous utiliserons comme types d'attributs, seront utilisées comme de simples types scalaires : les objets de ces classes n'auront pas d'existence propres, en fait ils se comportent exactement comme des **valeurs**[1]. Ils ne seront pas pointés, il ne seront pas partagés, et si 2 objets entités (2 Astres par exemple) se trouvent avoir la même couleur et le même rayon il s'agit bien des même couples de valeurs dupliqués en 2 exemplaires en mémoire. Le C++ est un langage orienté objet particulièrement bien adapté aux « types valeurs » puisque la sémantique par défaut des attributs objets, des passages de paramètres objets et des affectations entre objets est une sémantique par valeur (la nouvelle valeur écrase l'ancienne). Dès que nous serons familiarisés avec les « types valeurs » d'une application nous pourrons les éliminer des schémas UML et ne plus les représenter graphiquement reliés par des associations de composition mais directement en tant qu'attributs :



Etape 1 :
Diagramme des classes des types valeur



Etape 2 :
Diagramme de classes des types entités

Un mot sur les identifiants. Trouver des noms appropriés pour les identifiants informatiques est une activité extrêmement sérieuse, ceci ne doit pas être pris à la légère et doit faire l'objet d'un temps de réflexion et de délibération si on est en équipe : l'identifiant informatique doit être sémantiquement cohérent avec ce qu'il désigne (Personne ≠ Identité) précis (Client ≠ Personne) et si possible court (Client ≠ PersonneClienteDuSysteme) d'autant plus qu'il est utilisé souvent et localement (int i) mais pas trop court non plus (ipc ≠ idxPersonneCree) et sans accents ce qui pose des problèmes avec les infinitifs en français (trouvé ≠ trouve). Tout ceci est d'autant plus important quand on identifie des types (càd le nom des classes) : **le nom de la classe est comme une promesse de ce que les objets de cette classes pourront faire**.

Et il faut aussi respecter les conventions suivantes : paramètres, variables locales et attributs commencent par une minuscule (avec m_ pour les attributs), les types commencent par une majuscule (sauf les types de la bibliothèque standard std::string etc...). Il est naturel d'avoir envie de s'approprier des énoncés qui sont très guidés, et les identifiants ressemblent à un des derniers espaces de liberté où on peut exprimer sa « créativité ». Mais jouer avec les identifiants des classes est sans doute une mauvaise idée.

1 Une des approche en conception objet brièvement abordée au cours 6, Domain Driven Design, distingue les **entités** qui ont une identité (Mercure, Vénus, Mars...) et les **objets valeur** qui n'ont pas d'identité propre (la position x=150, y=30, l'apparence rayon=20 couleur="green").

Ces réflexions et analyses papier initiales peuvent sembler excessive et vous pouvez vous demander quand-est qu'on commence à coder... Après tout un bon développeur n'est-il pas rémunéré pour taper sur des touches de clavier dans un éditeur de code source ? Sans doute un développeur senior peut aborder un projet simple comme celui-ci en attaquant directement le code, mais c'est bien parce qu'il est capable de mener ces réflexions « dans sa tête » en amont du code. Vous n'imaginez pas ce qu'une bonne réflexion initiale peut faire gagner à un projet logiciel, ni ce qu'une mauvaise réflexion peut lui faire perdre. C'est la raison d'être de l'ingénieur ou architecte logiciel : proposer de bons modèles objets en amont du code.

Un défaut du modèle objet proposé : le centre de gravité obtenu par le calcul sera toujours stocké dans un type Astre. **Mais que faire avec les attributs d'aspect pour un tel Astre « centre de gravité » ? Dans la suite je propose de "neutraliser" ses attributs en leur donnant des valeurs par défaut (rayon nul, couleur "black") comme au TP précédent.**

Ce n'est pas sémantiquement satisfaisant, mais la façon correcte d'aborder ça passerait par le modèle plus complexe en bas de la page 2 et par l'héritage et le polymorphismes d'une classe abstraite Aspect, techniques pas encore abordées. Ou alors il faudrait revoir complètement le modèle : avec Barycentre en classe mère et Astre en classe fille qui ajoute un Aspect.

*1.4 Faites le diagramme de classes UML correspondant à une version héritage de ce qui est proposé avec une composition sur le diagramme d'objets en bas de la page 2 (remplacer la composition par un héritage « équivalent »). L'objectif serait d'utiliser le type Barycentre pour stocker les objets « centre de gravité ». Discussion : est-ce que dire « un Astre **est** un Barycentre » vous semble sémantiquement valable ? Et dire « un Astre **a** un Barycentre » ? Et si au lieu d'appeler la classe mère Barycentre on l'appelle CentreDeMasse ? Conclusion(s) ?*

2. Méthodes : ce que ces données vont faire pour nous !

*Au fait on veut **faire quoi** avec cette application ? On n'a encore pas parlé des méthodes et il n'y a pas une seule méthode dans les diagrammes de classes proposés ! Quelles sont les actions proposées associées à ces données si soigneusement structurées ?*

Pour rappel l'application doit nous permettre d'avoir un menu interactif qui propose :
- ajouter un astre (saisir ses paramètres) -> new
- dessiner la situation actuelle dans un nouveau fichier output.svg (écraser l'ancien)
dans le dessin on peut voir tous les astres en place, et leur centre de gravité indiqué
- afficher tous les astres (leurs attributs) à la console, numérotés
- enlever un astre en saisissant son numéro -> delete
- quitter

On peut faire une « analyse descendante » pour connaître les besoins en méthodes et opérateurs des objets : partant des opérations (méthodes) des classes composites quels seront les opérations de classes composantes ?

Ne serait-ce que pour tester, il va falloir pouvoir construire des objets avec des valeur initiales pour tous les attributs, et ce pour toutes les classes. Souvent (mais ce n'est pas systématique) le client de la classe composite préférera donner tous les attributs sous forme de liste non structurée. Avec la surcharge on pourra proposer les 2 formes, au choix, au code client.

```
// Déclaration d'un objet de type Astre avec liste "non structurée"
Astre terre{ 200, 400, 6, 63, "blueball" };

// Déclaration d'un objet de type Astre avec liste de composants
Astre terre{ {200, 400}, 6, {63, "blueball"} };

// Cette 2ème forme est équivalente à ce code avec intermédiaires
Coords positionTerre{ 200, 400 };
AspectSphere aspectTerre{ 63, "blueball" };
Astre terre{ positionTerre, 6, aspectTerre };
```

La construction avec paramètres exhaustifs pour les objets de la classe composite Astre implique qu'on doit aussi avoir des constructeurs avec paramètres exhaustifs pour les objets des classes composantes Coords et AspectSphere.

Le CDC implique qu'on doit pouvoir saisir les données d'un Astre. On peut faire ça en amont de la création de l'objet, au niveau du code client, et injecter les valeurs saisies dans l'appel au constructeur paramétré. Ou alors on peut prévoir une méthode saisir dans chaque classe. Dans ce cas les classes doivent disposer d'un constructeur par défaut. Ce constructeur par défaut est appelé par le client pour créer un objet « vide » c'est à dire un objet avec des valeurs par défaut, puis la méthode de saisie est appelée et les attributs sont renseignés à l'intérieur de la classe. Ça permet au code client de déléguer les détails des opérations de saisie aux méthodes des classes concernées[2].

Un objet Astre devra pouvoir être affiché en console : méthode afficher pour Astre et pour ces composantes. Un objet Astre devra pouvoir être dessiné dans output.svg : méthode dessiner pour Astre en utilisant ces composantes (besoin probable d'accesseurs en lecture). Un objet Astre devra pouvoir être détruit, mais le cours indique qu'il n'y a pas besoin de destructeur pour des classes avec des attributs qui ont des sémantiques par valeur ce qui est le cas ici. Destructeurs pas absolument indispensables donc, mais on les mettra quand même pour vérifier qu'on comprend bien le cycle de vie des objets (là encore, dans le composite et dans ses composantes).

Enfin le cœur du sujet, le calcul de centre de gravité, se fera par une méthode Astre::sommer qui prend en paramètre un 2ème objet Astre (le 1er étant l'objet cible `this`) et qui retourne un Astre représentant le centre de gravité. Cet Astre « centre de gravité » pourra à nouveau être sommé etc... jusqu'à obtention de la « somme » (le barycentre) de tous les Astres du système. On voudra rapidement améliorer le confort au niveau du code appelant en surchargeant l'opérateur operator+ dans sa version symétrique (nous y reviendrons) qui n'est pas une méthode et ne pourra donc pas accéder aux attributs private. Mais plutôt que de mettre en place moult mutateurs inutiles sur les attributs on préférera donner à `Astre operator+(const Astre&, const Astre&)` qui n'est pas une méthode une intimité avec la classe Astre en la déclarant amie (friend). Des indications seront fournies.

2 Entre la déclaration de l'objet initialement « vide » et l'appel à la méthode de saisie l'objet existe dans un état « non utile » ou « non utilisable » ce qui est généralement à éviter. Cependant les alternatives semblent pour l'instant trop compliquées : on pourrait par exemple prévoir un constructeur qui reçoit une (référence) à un flot std::istream et qui va chercher les données initiales de l'objet dans ce flot, qui peut correspondre aussi bien à un fichier qu'à une saisie console si l'appelant passe std::cin en paramètre du constructeur. Ceci sera évoqué au cours 10, slide 66.

Et pour que le calcul proprement dit du barycentre (dans Astre::sommer ou dans operator+) s'exprime dans un langage avec un haut niveau d'abstraction, sans entrer dans des détails de bas niveau comme par exemple est-ce qu'on est en 2D ou en 3D, il faudra que la classe Coords mette à notre disposition 2 opérateurs de calcul vectoriel, somme vectorielle et multiplication par un réel (voir Cours 4 surcharge d'opérateurs qui est aussi bien valable pour une classe Coords que pour une struct à condition de déclarer ces opérations amies de la classe Coords)

2. Reprenez les 2 diagrammes de classes UML de la page 4 en les complétant par les méthodes.

Maintenant nous pouvons coder en partant du bas (les composants) vers le haut (le composite) !

3. Mise en place d'une classe Coords

Je vous suggère de développer la classe Coords dans un 1er temps dans un projet à part, puis d'intégrer les fichiers coords.h et coords.cpp au projet barycentres seulement quand la classe Coords est au point. Le projet peut s'appeler coords (faites simple!).

Développez une classe Coords qui regroupe 2 réels x et y (utiliser double, plus précis que float pour les simulation scientifiques ou techniques) en respectant les conventions et consignes données en cours. Mettre en place un constructeur à 2 paramètres x et y qui initialise les 2 attributs. Tester. Comme les attributs sont privés on ne peut pas « voir » ce qui se passe dans la classe depuis le code client, ***immédiatement le réflexe est de se donner de la visibilité sur ce qu'on manipule*** *: mettre en place une méthode afficher qui affiche les attributs en console. Tester : vous pouvez déclarer 2 objets Coords avec des valeurs initiales (quelconques, par exemple 20, 30 et 70, 80) et afficher ces 2 objets depuis le code client (le main) en appelant la méthode afficher.*

Grâce à la méthode afficher vous avez de la visibilité sur les objets Coords créés : vous allez pouvoir enrichir la classe. Développez un constructeur par défaut (sans paramètre) qui quand il est utilisé (un objet est déclaré sans paramètre) construit un objet Coords avec des valeurs par défaut. En général pour des attributs numériques on choisit simplement 0. Il est possible de ***déléguer*** *l'initialisation par défaut au constructeur paramétré : voir slide 86 du cours 5.*

On pourra remplir ces objets par défaut avec un appel à une méthode de saisie : demander à l'utilisateur d'entrer les coordonnées avec std::cin. On ne demande pas de « blindage » à ce niveau mais vous aurez la politesse d'afficher à l'utilisateur un message lui indiquant ce qu'on veut de lui par exemple : « veuillez entrer x et y SVP : ». Tester : dans le main déclarer 2 objets sans paramètres et appeler leur méthode de saisie puis leur méthode d'affichage.

Pour des raisons d'apprentissage (pas par nécessité technique) nous allons vouloir comprendre le cycle de vie des objets en définissant un destructeur. Dans ce destructeur nous afficherons en console que nous détruisons un objet de type Coords, et les valeurs de cet objet. Bien sûr on ne veut pas re-coder dans le destructeur les std::cout des attributs : on appelle la méthode afficher. Tester avec 2 objets déclarés dans le main, l'un étant initialisé avec des valeurs «en dur» et l'autre saisi. Afficher un message « Fin du main » juste avant le return 0; qui termine le main : les destructeurs sont-ils appelés avant ou après ce message ? Est-ce cohérent avec le cycle de vie des objets automatique : voir slide 66 du cours 5. Pouvez vous prévoir ce qui va se passer avec un objet déclaré dans un bloc { } ? Tester :

```
for (int i=0; i<10; ++i)
{
    Coords c;
    c.afficher();
}
```

Implémenter les 2 opérateurs de calcul vectoriel : somme vectorielle et multiplication par un réel (voir Cours 4 surcharge d'opérateurs). Compiler. Que se passe-t-il ? Pour donner un accès aux membres privés nous pouvons définir des accesseurs publiques getX et getY, ou si nous considérons que les opérateurs doivent être aussi intimes avec la classe que si ils étaient des méthodes nous pouvons déclarer ces opérateurs ***amis de la classe*** *: dans coords.h,* ***à l'intérieur de la classe****, par exemple à la fin de la déclaration des méthodes publiques, ajouter les déclarations d'amitié :*

```
friend Coords operator+(const Coords& c1, const Coords& c2);
friend Coords operator*(double m, const Coords& c);
```

Tester en déclarant 2 objets Coords dans le main et en affichant le résultat de calculs vectoriels avec eux. Tester un calcul ***manuel*** *de barycentre de 2 objets Coords a et b qui auraient comme poids 3 et 2 en affichant le résultat du calcul* `(1.0/(3+2))*(3*a+2*b)`*. La notation* `(3*a+2*b)/(3+2)` *marcherait-elle ? Pourquoi ? Comment faire ? Faite le !*

Enfin pour injecter les données séparément x et y dans les méthodes de la classe Svgfile (qui ne connaît pas le type Coords : void Svgfile::addDisk(double x, double y, …) il faut prévoir deux accesseurs en lecture getX et getY. Implémentez les et testez les depuis le main en affichant séparément le x et le y d'un objet Coords.

4. Mise en place d'une classe AspectSphere

Cette classe est moins excitante que la précédente... Si vous débutez en C++ / POO c'est votre 2ème classe et vous allez tout de suite vous rendre compte que le développement objet n'est pas dénué de répétition, voir parfois franchement fastidieux (boilerplate code : beaucoup de tuyaux à brancher). Suivez un protocole de développement similaire pour implémenter et tester cette classe utilitaire. Bien sûr dans le cas de la classe AspectSphere il n'y a pas d'opérateurs à surcharger parce que sommer ou multiplier des objets AspectSphere n'aurait pas beaucoup de sens ni d'utilité.

5. Mise en place de la classe Astre dans le projet définitif

Vous pourrez vous ***inspirer*** *du code du TD/TP 4 pour faciliter l'intégration des classes Coords et AspectSphere a la classe Astre mais* ***il est fortement recommandé de ne pas essayer de repartir du projet du TD/TP 4 et de le transformer*** *: repartez avec un nouveau projet initial propre. Il est possible d'avoir plusieurs projets ouverts dans CodeBlocks mais ne vous embrouillez pas !*

Re-téléchargez le projet de départ qui contient la « bibliothèque » de dessin vectoriel

https://fercoq.bitbucket.io/cpp/tdtp/tdtp4/barycentres_exo.zip

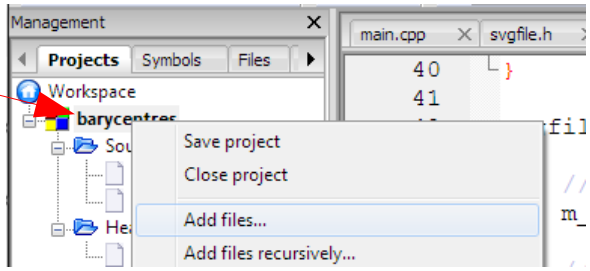
Dézippez, ouvrez le projet (.cbp) si vous utilisez Code::Blocks ou faites un projet qui inclut les 3 fichiers sources sinon, compilez, exécutez. Vous devez obtenir dans le répertoire d'exécution[3] (macOS : voir note de bas de page) un nouveau fichier ***output.svg*** *qui a été généré par l'exécution.*

C'est le même projet de départ que pour le TD/TP 4, vous devriez retrouvez vos marques ! Ajoutez au projet les fichiers coords.h et coords.cpp et aspectsphere.h et aspectsphere.cpp : pour CodeBlocks il suffit de ***copier*** *ces fichier source dans le répertoire du nouveau projet barycentres, puis dans CodeBlocks de faire un* ***clic droit*** *sur le nom du projet et Add files... Ensuite sélection*

3 Pour Code::Blocks c'est le répertoire de projet. Avec d'autres systèmes ça peut être ailleurs. Pour macOS avec Xcode c'est remarquablement difficile à trouver, Google « xcode console execution directory » -> ici ou là

multiple (Maintenir Ctrl) des 4 fichiers cible, puis Ouvrir et OK. Vérifier que ça compile, inclure les .h, vérifier qu'au niveau du main on peut utiliser des objets des classes Coords et AspectSphere.

clic droit



Créer 2 nouveaux fichiers (l'interface .h et l'implémentation .cpp) pour la classe Astre. Déclarer la classe composante Astre en utilisant les types composites... Vérifier que ça compile (avez-vous bien inclus les .h des classes composites dans astre.h?). Compléter la classe Astre avec un constructeur avec tous les paramètres. Voir chapitre H du Cours 5 en particulier slide 121, la syntaxe par liste d'initialisation est recommandée. Tester le code suivant :

```
Astre terre{ 200, 400, 6, 63, "blueball" };
```

Ça doit compiler, mais on ne sait pas si les valeurs sont bien en place. ***Immédiatement le réflexe est de se donner de la visibilité sur ce qu'on manipule*** *: mettre en place une méthode* ***Astre::afficher*** *qui affiche les attributs en console.* ***Pour les attributs composants on voudra éviter de passer par les accesseurs en lecture et on préférera appeler les méthodes afficher de ces composants****. Si elles font des retours ligne en trop vous pouvez modifier ces méthodes Coords::afficher et AspectSphere::afficher en enlevant des std::endl, ou optionnellement ajouter un paramètre booléen retourLigne, avec valeur par défaut, permettant à l'utilisateur de la méthode de choisir si il veut les retours ligne. Tester.*

Ajouter un constructeur par défaut pour Astre (penser à déléguer au constructeur paramétré). Tester en déclarant un objet Astre sans paramètre et en l'affichant. Ajouter une méthode de saisie dans la classe Astre. ***Comme pour l'affichage on délègue aux composants : la saisie des attributs composants se fera en appelant la méthode de saisie des composants !*** *Tester en appelant la méthode saisie sur un objet Astre construit par défaut, puis afficher le résultat.*

Comme pour les classes composantes, dotez la classe composite d'un destructeur explicite qui affiche (les données de l') objet détruit. Tester : la destruction d'un objet composite implique-t-il bien la destruction de ces composants ?

Développer la méthode Astre::dessiner qui prend en paramètre une référence à un Svgfile et qui ajoute le disque correspondant à l'astre dans le dessin si le rayon n'est pas nul, ou bien dessine une croix sinon (représentation d'un objet « centre de gravité »). L'utilisation des accesseurs en lecture des attributs des composants est nécessaire. Tester avec le système :

```
Astre terre{ 200, 400, 6, 63, "blueball" };
Astre lune{ 584, 400, 2.7,  27, "greyball"};
Astre cgtest{400, 400, 10, 0, "black"};
```

Développer la méthode Astre::sommer qui prend en paramètre un (référence constante à) 2ème objet Astre (le 1er étant l'objet cible `this`*) et qui retourne un Astre représentant le centre de gravité des 2. Bien entendu dans cette méthode on veut utiliser les opérateurs de calcul vectoriel proposés par le composant position de type Coords : on fera le calcul selon l'une des notation en haut de la page 8 en utilisant directement / et * et + de manière appropriée entre réels (type double) et vecteurs (type Coords). Tester (code page suivante) :*

```
Astre cgtest = terre.sommer(lune);
cgtest.dessiner(svgout) ;
```

Cette syntaxe n'est pas assez symétrique au goût du codeur client : en effet pourquoi la terre devrait-elle avoir cette position d'objet cible alors que la lune ne serait qu'un vulgaire paramètre ? Implémenter et tester une surcharge de l'opérateur operator+ entre Astres. Il n'est pas indispensable de le rendre ami de la classe Astre pour qu'il soit intime des attributs privés car il pourra faire appel à la méthode sommer ! Il est juste prototypé en dehors de la classe dans astre.h:

```
Astre operator+(const Astre& a1, const Astre& a2);
```

De cette façon le client doit pouvoir écrire :

```
Astre cgtest = terre + lune;
cgtest.dessiner(svgout);
```

6. Collection de Spheres : allocation dynamique

On reprend ici à peu près le même énoncé que l'exo 7 du TD/TP 4 mais cette fois ci avec une classe Astre au lieu d'une struct Sphere. On veut pouvoir ajouter/enlever des astres à une collection, de façon interactive. Les objets astres sont à la fois suffisamment légers et isolés (il n'y a pas d'autres objets qui pointent sur un objet Astre) pour pouvoir être traités par valeur : on pourrait gérer une collection d'astres sous forme d'un vecteur d'Astres std::vector<Astre> directement, sans allocation dynamique, de la même façon qu'on peut gérer interactivement une collection d'entier std::vector<int>.

Mais quand on commencera à avoir des objets plus gros (en octets) et surtout qui sont référencés (pointés) par d'autres objets cette approche posera problème, par exemple les vecteurs grossissent automatiquement (push_back...) mais ça implique de bouger en mémoire les données stockées, une donnée pointée devient alors invalide ... Il devient rapidement nécessaire de faire de l'allocation dynamique et de stocker les collections non pas sous la forme « objets valeurs » mais sous la forme « pointeurs sur entités stables ».

On aura donc un stockage de type std::vector<Astre*>. Le but de cet exercice est de s'entraîner à traiter ce genre de collection. Le *slide 108 du cours 4* montre le principe général, avec des sous-programmes de gestion de la collection qui prendront en paramètre une **référence à un tableau de pointeurs sur les objets**. Les méthodes valables pour traiter un objet à la fois trouveront une correspondance avec des sous-programmes acceptant ce type de vecteur. Ces sous-programmes ne sont pas des méthodes de la classe Astre (une méthode gère un seul objet à la fois!) mais comme ils manipulent des Astres ils seront prototypés dans astre.h et codés dans astre.cpp.

```
Astre Astre::sommer(const Astre& s) const;     // Méthode 1 seul
Astre sommer(const std::vector<Astre*>& vec); // Sous-prog plusieurs
```

Idem pour afficher (à la console) et dessiner (sur svgout…). Les sous-progs qui prennent la collection d'objets en paramètre appelleront les méthodes qui traitent un objet à la fois. Penser au const pour les paramètres invariants et les méthodes qui n'altèrent pas l'objet cible.

Implémenter tout ça : arriver à l'application telle que décrite en bas de la page 5 (menu...)

Pour effacer le $i^{ème}$ élément d'un vecteur vec : `vec.erase(vec.begin()+i);`
Attention : le pointeur est effacé, pas l'objet pointé. Appeler `delete vec[i];` ***avant*** *!*